1988-10

The Design of Software Interfaces in Spec

Berzins, Valdis

IEEE

V. Berzins, "The design of software interfaces in Spec," Proceedings. 1988 International Conference on Computer Languages, Miami Beach, FL, USA, 1988, pp. 266-270.
http://hdl.handle.net/10945/64259

The Design of Software Interfaces in Spec

Valdis Berzins

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

This paper presents a language for giving black-box specifications in the early stages of software design. The underlying computational model combines message passing with temporal events in a precisely defined way. The features of the language, especially those important for large scale design are presented by means of examples.

Keywords Black-box specifications, abstractions, specification language, computer aided software engineering, distributed systems, real-time systems.

1. Introduction

Spec is a formal language for writing black-box specifications for components of software systems. Black-box specifications are essential for realizing the benefits of abstractions in the software development process [2]. The critical early stages of software development are dominated by the tasks of building conceptual models of the proposed software and defining its interfaces. The Spec language is used in the functional specification stage for recording black-box specifications of the external interfaces of the proposed system, and in the architectural design stage for recording black-box specifications of the internal interfaces of the proposed system.

A formal specification language such as Spec is needed for defining the desired behavior of the proposed system before it is built, because English and other informal notations are too imprecise. Precision is important because in a large project many people have to agree on the interpretation of the specifications to produce a correct implementation. Written specifications are attractive as a communications medium in very large projects because the effort of writing a formal specification is independent of the number of people reading it, whereas communications overhead tends to increase with the size of the project in more informal techniques. Formal notation is important because it enables mechanical processing, opening the way to higher levels of computer-aided design than are currently used in software development [4]. Programming languages such as Ada are formal, but are not well suited for writing black-box specifications because they have been designed for describing the algorithms and data structures realizing a module rather than the behavior a module presents at its interface.

There has been much previous work on providing programming language support for abstractions [8, 10, 14, 19, 23]. Much of the previous work on formal specifications has focused on the problem of proving the correctness of programs [9, 11, 16, 21, 25]. Spec has been intended primarily for supporting the use of abstractions in the design of software systems. Surveys of related work can be found in [7, 24]. Spec has evolved from an earlier specification language [1] and a rapid prototyping language for the design of large real-time systems [20], guided by extensive classroom experience in using formal specifications in multi-person projects [2]. The most important advances over the earlier language are the integration of time into the underlying model, the development of an inheritance mechanism [3], and the separation of granularity and control state considerations from the event-level interfaces of a module. The Spec language is suitable for specifying parallel, distributed, or time sensitive systems as well as conventional systems.

Spec differs from algebraic specification languages such as Larch [12, 13] because it is based on models rather than theories. While it is feasible to write Spec axioms in the conditional equation form commonly used in algebraic approaches, the use of models and axioms of other forms can sometimes lead to simpler specifications. The restricted form of Larch is helpful for supporting automated tools for program verification, while the expressiveness of Spec is useful in developing large scale designs. Larch is based on the premise that interfaces involving state changes are inherently dependent on the implementation language. Larch provides general purpose facilities for defining immutable data types along with a framework for adding an implementation-language dependent layer for defining state changes and concrete interfaces. Spec is based on the premise that interfaces with state changes, exceptions, concurrent interactions, and time dependencies can all be specified independently of implementation language, and that the definition of a language dependent concrete interface is a matter of packaging rather than semantics. This reflects the difference between the prescriptive nature of specifications used as a design tool and the descriptive nature of specifications used primarily to prove properties about systems.

Model based approaches such as VDM [6] have a few similarities to Spec. However, Spec has been designed to handle systems with a wide range of features, e.g. concurrency and time dependent constraints, while VDM is primarily intended for specifying sequential systems [7].

The GIST language is based on a global state model approach that describes behavior independently of interfaces [17], and is intended for use in the early stages of requirements analysis where properties of the entire application are being determined without assigning boundaries or allocating functions to either the proposed software system or its environment. Spec makes no attempt to address this stage, and is intended for use in the later functional specification and architectural design stages, where properties of proposed external and internal interfaces are specified. Localization of information, treatment of distributed systems, and treatment of real-time constraints are explicit design goals of Spec. The goals of Spec and GIST are complementary and the two can be used together at different phases of software development.

Spec is based on the event model of computation, and uses predicate logic for the precise definition of the desired behavior of modules. The most important ideas of this language are modules, messages, events, atomic transactions, and defined concepts. Events can be used for defining timing constraints, while localized states and atomic transactions are important for specifying distributed or concurrent systems. Spec supports reuse of abstractions via inheritance and generic modules. Spec also has features important for specifying large conventional systems, such as import/export controls for defined concepts, and view and inheritance mechanisms.

2. The Event Model

The Spec language uses the event model to define the behavior of black box software modules. The event model has been influenced by the actor model [15, 26]. The main differences from the actor model are the treatment of time and temporal events, and the treatment of multi-event transactions [1]. In the event model, computations are described in terms of modules, messages, and events. A module is a black box that interacts with other modules only by sending and receiving messages. A message is a data packet that is sent from one module to another. An event occurs when

a message is received by a module at a particular instant of time.

Modules can be used to model external systems such as users and peripheral hardware devices, as well as software components. Modules are active black boxes, which have no visible internal structure. The behavior of a module is specified by describing its interface. The interface of a module consists of the set of stimuli it recognizes and the associated responses. A stimulus is an event, and the response is the set of events directly triggered by the stimulus. The events in the response consist of the arrivals of the messages sent out by the module because of the stimulus.

Messages can be used to model user commands and system responses. Messages represent abstract interactions that can be realized in a wide variety of ways, including procedure call, return from a procedure, Ada rendezvous, coroutine invocation, external I/O, assignments to non-local variables, hardware interrupts, and exceptions. A message has a condition, a name, and a sequence of zero or more data values. The condition has the value **normal** for messages representing normal interactions, and the value **exception** for messages representing abnormal interactions such as exceptions. The name of a message identifies the service requested by a normal message or the exception condition announced by an exception message. The data values represent either inputs or results, and may be present for any kind of message. The triggering event is an implicit attribute of each message, used for identifying the destination for reply messages.

Each module has its own local clock and can send messages to itself at times determined by its local clock. The arrival of such a message is called a **temporal event**. Temporal events allow modules to initiate actions as well as responding to external stimuli.

Events at the same module happen one at a time, in a well-defined order. This order can be observed as a computation proceeds, and corresponds to the ordering of the local times at which those events occur. Events at different places need not have a well-defined order because the local clocks of different modules are not guaranteed to be precisely synchronized with each other. This is realistic since perfect synchronization of clocks at different locations is not possible in practice. Clocks can be synchronized only by sending messages with a non-zero delay. Clock synchronization depends on unverifiable assumptions about the unknown message delays involved (e.g. delays in both directions are equal) and the relative rates of the local clocks.

Because there is no completely accurate global time reference, the only guaranteed orderings between events are derivable from discrete sequences of the following types of steps:

(1) Two events at the same module are ordered by their local times.

(2) The event which triggered the sending of a message comes before the event in which that same message is received.

Message transmission is assumed to be reliable, which means every message sent eventually arrives at its destination. In the absence of explicit specifications constraining the delay, messages can have arbitrarily long and unpredictable transmission delays. Specifications for message delays are inherently approximate unless the origin and destination of the message are the same.

The response of a module to a message is influenced only by the sequence and arrival times of the messages received by the module since it was created. This means there is no action at a distance: all interactions must involve explicit message transmissions. This restriction is a formalization of the requirement that each module must correspond to a coherent abstraction.

The event model and the Spec language admit nondeterminism due to partially specified communication delays or partially specified responses. Complete specifications admit only deterministic behavior. In Spec it is possible to specify that a response must be deterministic (repeatable) without completely specifying the other properties of the response.

Each module has the potential of acting independently, so that there is natural concurrency in a system consisting of many modules. Since events happen instantaneously and the response of a module is not sensitive to anything but the sequence of events at the module, the event model implies concurrent interactions with a module cannot interfere with each other at the level of individual events. Atomic transactions can be used to specify constraints on the order in which a module can accept events. Atomic transactions can be used to specify synchronization constraints involving chains of events in distributed systems. Atomic transactions must be used with care, because they can interact with each other or with timing constraints to produce unsatisfiable specifications. Deadlocks are a well known example of such situations.

Modules can be used to model concurrent and distributed systems, as well as systems consisting of a single sequential process. The event model helps to expose the parallelism inherent in a problem, since a stimulus can have a set of unordered responses occurring at different locations.

3. Specifying Software in Terms of Events

The Spec language provides a means for specifying the behavior of three different types of modules: functions, state machines, and abstract data types. Messages can also be used to model generators or iterators [18,22]. The properties of these kinds of modules and message are described below, with examples of each.

3.1. Functions

The response of a function module is influenced only by the most recent stimulus, so that function modules do not exhibit internal memory. Completely specified function modules calculate single-valued functions in the mathematical sense. An example of a specification for a square_root function is shown below.

```
FUNCTION square_root {precision: real} WHERE precision > 0.0

  MESSAGE(x: real)
    WHEN x >= 0.0
      REPLY(y: real)
      WHERE y >= 0.0 & approximates(y * y, x)
    OTHERWISE REPLY EXCEPTION imaginary_square_root
  CONCEPT approximates(r1 r2: real)
    -- True if r1 is a sufficiently accurate approximation of r2.
    -- The precision is relative rather than absolute.
    VALUE(b: boolean)
      WHERE b <=> abs((r1 - r2) / r2) <= precision
END
```

Function modules usually provide only a single service, and are usually designed to accept anonymous messages. The square_root function accepts anonymous messages containing a single real number. The response of a module to a message can be defined with several cases introduced by WHEN clauses. The predicate after each WHEN is a precondition, describing the conditions under which the associated response must be triggered by an incoming message with a given name and condition. The preconditions in each WHEN statement are stated independently, so that the order of the WHEN statements does not matter.

OTHERWISE is an abbreviation for the case where none of the other WHEN statements apply. In the example above, the OTHERWISE means the same thing as WHEN x < 0.0. In the Spec language each series of WHEN statements must be terminated by an OTHERWISE, to make sure all cases are covered. If a case is to be left undefined, the designer must say so explicitly.

A REPLY describes the message sent back in response to a stimulus. The reply is sent to the module originating the message that arrived in the stimulus, determined by the implicit origin attribute of the message. If REPLY is followed by EXCEPTION then the condition of the reply message is **exception**, representing an exceptional event, and otherwise the condition of the reply message is **normal**, representing a normal response. EXCEPTION can also appear after MESSAGE in the specification of an exception handler, indicating that the stimulus must be an exception condition.

An outgoing message such as a REPLY can have a WHERE clause, which describes a postcondition that must be satisfied by the outgoing message. The WHERE keyword is followed by a statement in predicate logic describing the relation between the contents of the message that was received and the contents of the reply message. This predicate states how to recognize a correct result, but it does not specify how to compute the required output.

Whenever a message arrives which matches a MESSAGE header of a module and satisfies the precondition (WHEN) of one of the cases, then a response must be sent which matches the REPLY header and satisfies the associated postconditions (WHERE). A message matches a header if the message has the specified name, condition, and number of data values, and

if each data value belongs to the specified data type. A message satisfies a predicate if the predicate is true in the state where the formal arguments of the visible message specifications are bound to the actual data values in the message. Only the incoming message is visible in a precondition, while the incoming message and all associated outgoing messages are visible in a postcondition.

Messages without any WHEN clauses have a single case whose precondition is always true. If the precondition for more than one case is satisfied, all of the associated responses must be sent and the constraints of all the associated postconditions must be met simultaneously. Overlapping preconditions are not recommended because they can lead to inconsistencies.

The concept *approximates* defines the intended meaning of "sufficiently accurate approximation" in terms of the generic parameter *precision*. The generic parameter allows a single template for a square root module to be adapted to many applications with different precision requirements. Some notion of approximation is needed to specify a practical square root function because it is not possible to implement exact square roots using machine arithmetic. In this case the size of the acceptable interval is defined relative to the size of the input value rather than as an absolute constant. Introducing an explicitly defined concept modularizes the specification. This helps simplify the postcondition and supports stepwise refinement and localization of information. The definition of the concept can be delayed or left as an informal comment when the concept is identified and the postcondition is developed.

3.2. Machines

A machine is a module with an internal state, i.e. machines are mutable modules. An example of a machine is shown below.

```
MACHINE inventory
  -- assumes that shipping and supplier are other modules
 STATE(stock: map[item, integer])
 INVARIANT ALL(i: item :: stock[i] >= 0)
 INITIALLY ALL(i: item :: stock[i] = 0)

 MESSAGE receive(i: item, q: integer)
   -- Process a shipment from a supplier.
  WHEN q > 0
   TRANSITION stock[i] = *stock[i] + q
    -- Delayed responses to backorders are not shown here.
  OTHERWISE REPLY EXCEPTION empty_shipment

 MESSAGE order(io: item, qo: integer)
   -- Process an order from a customer.
  WHEN 0 < qo <= stock[io]
   SEND ship(is: item, qs: integer) TO shipping
    WHERE is = io, qs = qo
   TRANSITION stock[io] = *stock[io] - qo
  WHEN 0 < qo > stock[io]
   SEND ship(is: item, qs: integer) TO shipping
    WHERE is = io, qs = stock[io]
   SEND back_order(ib: item, qb: integer) TO supplier
    WHERE ib = io, qb = qs - qo
   TRANSITION stock[io] = 0
  OTHERWISE REPLY EXCEPTION empty_order
END
```

The behavior of a machine is described in terms of a conceptual model of its state, rather than directly in terms of the messages that arrived in the past, because such descriptions are usually shorter and easier to understand. The components of the conceptual model of the state are declared after the keyword STATE, and restrictions on the set of meaningful states are given after the keyword INVARIANT. Restrictions on the initial state are given after the keyword INITIALLY. The restrictions after INVARIANT must be satisfied in all reachable states, while the restrictions after INITIALLY must be satisfied only in the first state.

State changes are described by predicates after the keyword TRANSITION. In such statements, plain variables of the form x refer to the value of x in the current state (just after the arrival of the stimulus), while variables of the form *x refer to the value of x in the previous state (just before the arrival of the stimulus). The transitions in the example are equations rather than assignment statements. Equations can describe the transition either forwards or backwards in time, whichever is simpler (cf. the first two transitions). The *x notation can only be used in the INVARIANT, the TRANSITIONS, and in WHERE clauses describing the output in terms of the new state. The Spec language follows the convention that components of the state of a machine or the model of an abstract data type do not change unless the component is explicitly mentioned in a TRANSITION clause.

The SEND statement is used instead of REPLY to describe messages sent to destinations other than the origin of the incoming message. A SEND statement means that a message satisfying the description must be sent to the given destination. SEND statements are useful for describing distributed systems with a pipeline structure. There can be only one REPLY, but there can be any number of SEND's. If there is more than one SEND, the message transmissions can be performed concurrently or one at a time in any order, without waiting for any responses. The example has such a multiple response to the *order* message in the case where there are not enough items on hand to fill the order completely.

3.3. Types

A type module defines an abstract data type. An abstract data type consists of a value set and a set of primitive operations involving the value set. In the event model, a type module manages the value set of an abstract data type, creating all of the values of the type and performing all of the primitive operations on those values. Each message accepted by the type module corresponds to one of the operations of the abstract data type. The messages of a type module usually have names, since abstract data types usually provide more than one operation.

A module is **mutable** if the response of the module to at least one message it accepts can depend on messages that arrived before the most recent incoming message. A module is **immutable** if the response of the module to every possible message is completely determined by the most recent message it has received. Mutable modules behave as if they had internal states or memory, while immutable modules behave like mathematical functions. A module is immutable if and only if it is not mutable. An example of a specification for an immutable abstract data type is shown below.

```
TYPE rational
 INHERIT equality[rational]
 MODEL(num den: integer)
 INVARIANT ALL(r: rational :: r.den ~= 0)

 MESSAGE ratio(num den: integer)
  WHEN den ~= 0
   REPLY(r: rational)
   WHERE r.num = num, r.den = den
  OTHERWISE REPLY EXCEPTION zero_denominator

 MESSAGE add(x y: rational) OPERATOR +
  REPLY(r: rational)
  WHERE r.num = x.num * y.den + y.num * x.den,
   r.den = x.den * y.den

 MESSAGE multiply(x y: rational) OPERATOR *
  REPLY(r: rational)
  WHERE r.num = x.num * y.num, r.den = x.den * y.den

 MESSAGE equal(x y: rational) OPERATOR =
  REPLY(b: boolean)
  WHERE b <=> (x.num * y.den = y.num * x.den)
END
```

Data types have conceptual models, which are used to visualize and describe the value set of the type. The conceptual model is used to specify the behavior of a type, and forms the mental picture of the type for the programmers who use the operations of the type. The conceptual model is chosen for clarity, and is usually different than the data structure used in the implementation. In case the data type must be re-implemented to improve performance, the data structure used in the implementation will change, but the conceptual model will not.

Each instance of the type can be represented as a tuple containing the data components declared after the MODEL keyword. The restrictions on the components of the model are described in the INVARIANT, which selects a subset of the tuple data type defined by the MODEL to serve as the

conceptual representation. The INVARIANT is a predicate that must be true for all meaningful conceptual representations.

The invariant on the conceptual representation should be adjusted to make the descriptions of the operations as simple as possible. The invariant on the conceptual representation does not involve the implementation data structure and does not restrict the designer's choice of implementations. The invariants on the implementation data structures will often be much more complicated than the conceptual invariants, because implementation invariants often determine efficiency. Most books on data structures are really about the art of choosing implementation invariants that enable efficient algorithms.

Inside the module defining an abstract data type, predicates describing the effects of the operations can be written in terms of the conceptual representation. Inside the module defining an abstract type instances of the type can be described as if they were tuples containing the components specified in the MODEL. The notation x.y can be used to refer to the y component of the abstract data value x. The specifications of other modules may describe the values of abstract types only in terms of the MESSAGEs it provides and the CONCEPTs it EXPORTs.

It is sometimes convenient to express complicated conditions as lists of independent constraints. The predicates after INVARIANT, WHEN, and WHERE can be lists of expressions separated by commas. A list of statements is true if and only if all of the statements in the list are true individually, so that in this context a comma means the same thing as &. The comma has a lower precedence than all of the other operators, so that it can be used to separate statements at the top level without need for parentheses.

An example of a definition for a mutable type is shown below.

```
TYPE queue{t: type}
 INHERIT mutable{queue}
  -- Inherit definitions of the concepts "new" and "defined".
 MODEL(e: sequence)
  -- The front of the queue is at the right end.
 INVARIANT true

  -- Any sequence is a valid model for a queue.

 MESSAGE create
   -- A newly created empty queue.
  REPLY(q: queue{t}) WHERE q.e = [ ]
  TRANSITION new(q)

 MESSAGE enqueue(x: t, q: queue{t})
   -- Add x to the back of the queue.
  TRANSITION q.e = append([x], *q.e)

 MESSAGE dequeue(q: queue{t})
   -- Remove and return the front element of the queue.
  WHEN not_empty(q)
   REPLY(x: t)
   TRANSITION *q.e = append(q.e, [x])
  OTHERWISE REPLY EXCEPTION queue_underflow

 MESSAGE not_empty(q: queue{t})
   -- True if q is not empty.
  REPLY(b: boolean) WHERE b <=> (q.e ~= [ ])
END
```

In mutable types the instances of the type have internal states, and operations are provided for changing the internal states of the instances. TRANSITION clauses are allowed in types as well as machines. A type is mutable if and only if it has a non-trivial TRANSITION clause (i.e. a TRANSITION that implies *x ~= x for some component x). Mutating operations, such as *enqueue* in the example above, are described using TRANSITION clauses.

Object identity is an important issue for mutable types because all of the program variables bound to the same mutable object will be affected if a state changing operation is applied to the object. A new object is guaranteed to be distinct from all objects defined in the previous state. The concepts *new* and *defined* are not part of the Spec language, but they are provided by a predefined generic module *mutable* whose instances can be inherited by any mutable type. Spec provides facilities for specifying mutable types because they are used for efficiency reasons in internal interfaces of many systems. We recommend avoiding mutable types in user interfaces.

3.4. Generators

A generator is a message that generates a sequence of values one at a time. An example of a specification for a generator is shown below.

```
FUNCTION primes
 IMPORT prime FROM nat
 IMPORT sorted FROM sequence{nat}

 MESSAGE(limit: nat)
  GENERATE(s: sequence{nat})
  WHERE increasing_order(s),
   ALL(i: nat :: i IN s <=> 1 <= i <= limit & prime(i))

 CONCEPT increasing_order(s: sequence{nat})
  VALUE(b: boolean)
  WHERE b <=> sorted{less_or_equal@nat}(s)

END
```

The "@" is used in Spec to determine the type of an overloaded operator or constant in places where it is not clear from the context. The GENERATE keyword means the same thing as a REPLY except that the result is a sequence whose elements are delivered one at a time rather than all at once. This means that the elements will be generated one at a time, and processed incrementally, rather than being generated all at once and returned in a single data structure containing all of the elements, as would be the case for a REPLY of type sequence. In a program a generator is used to control a data driven loop. Generators can also be used in specifications of other modules, for example to define the range of a quantified variable. Generators are interpreted as sequence-valued functions when they appear in specifications.

Any message with a GENERATE is a generator, so that generators can be defined as operations of an abstract data type or a machine. This is an important application of generators, because it is otherwise difficult to scan all of the elements of an abstract collection without exposing the data structure used to implement the collection.

4. Features for Specifying Large Systems

The Spec language contains a number of features that are needed mostly for specifying large systems. Some of these features include generic modules, defined concepts, and an inheritance mechanism. An example illustrating the development of a complete system using Spec and a more detailed description of the language can be found in [5].

4.1. Generic Modules

A parametrized module specifies a family of modules rather than an individual module. Generic modules are important for achieving re-use of specifications and designs because they can be adapted to a wider variety of applications than their more specific instances. A parametrized module looks like an ordinary module definition except that there can be parameters after the module name, with an optional WHERE clause restricting the values of the parameters. The specifications for *square_root* and *queue* given in the previous section are examples of parametrized modules. Such a definition defines one module for each legal set of values for the parameters of the module. The parameters can range over data values values, functions or types.

4.2. Concepts

Concepts are important for explaining and testing the behavior of modules, and should be reflected in reference manuals and test oracles. A concept in the Spec language is a constant symbol, predicate symbol, or function symbol that can be used in constructing the logical assertions defining the behavior of modules. Concepts without formal arguments are interpreted as constants. A constant can be either a symbolic name for a data value or a symbolic name for a data type. Concepts with formal arguments are interpreted as predicate symbols if they have one VALUE and its type is boolean, and as function symbols otherwise.

Every concept is attached to some module, and is local to that module unless it is exported or inherited. Only concepts can be exported. If a con-

cept is exported, then it can be explicitly imported by other modules and used in their definitions. The export/import mechanism is used to record logical dependencies between modules, so that mechanical aid can be provided for tracing the impact of a proposed change to a definition.

A facility for introducing named concepts with explicit definitions and interfaces is important for organizing and simplifying descriptions of complex software systems. It is not a good idea to express a complicated constraint as a single very long expression in predicate logic, just as it is not a good idea to implement a large system as a single monolithic module: the result is too difficult for people to understand. Concepts have the same purpose in a specification language that subprograms do in a programming language, namely to provide a mechanism for orderly decomposition.

Concepts can also be used to mix formal and informal specifications, by a formal definition of a precondition, postcondition, invariant, or transition in terms of some concepts, and then providing informal definitions for the concepts. The formal definitions of the concepts can be filled in later, when the design has stabilized, or can be left out entirely if the details are not critical. The ability to mix formal and informal specifications in a disciplined manner can be important in practical projects with tight schedules.

Concepts represent the properties of the software that are needed to explain or describe the intended behavior of the software system. Concepts are delivered to the customer in the manuals explaining how the system is supposed to operate, where they may be explained less formally than in the functional specifications and architectural design. Concepts do not normally represent components of the code to be delivered, although it may be useful to implement them for testing purposes.

A function should be defined as a module of type FUNCTION if it is part of the model of the software system, and it should be defined as a concept that is part of a module if the function is needed to specify the behavior of the module, but is not part of the model of the system at the current level of description. If a function is needed to specify the behavior of a module at a high level of the architectural design, and is also one of the components used to realize that module at a lower level, then it should be defined as a concept attached to the module at the higher level and exported. At the lower level it should be specified as a FUNCTION module, which imports the concept from the higher level module and has a trivial definition in terms of the imported concept.

4.3. Views and Inheritance

The Spec language has an inheritance mechanism which can be used for specifying constraints common to the interfaces of many modules and for view integration. Specifying constraints common to many interfaces is essential for achieving interface consistency in very large systems. The interface of a system to each class of users can be a separate view of the system, perhaps specified by different designers. A total picture of the system is formed by expanding the definition of a module that inherits all of the individual views. The inheritance mechanism and the rules for combining different versions of messages and concepts inherited from multiple parents are described in more detail in [3].

5. Conclusions

Spec is a specification language with a broad range of applications. The language is primarily intended for recording black box interface specifications in the early stages of design. The language has a precise semantics and a simple underlying model. Experience has shown that it is sufficiently powerful to allow the specification of many kinds of software systems, and sufficiently flexible to allow software designers to express their thoughts without forcing them into a restrictive framework. The language is sufficiently formal to support mechanical processing. Some tools for computer-aided design of software that are currently under investigation are syntax-directed editors, consistency checkers, design completion tools, test case generators, and prototype generators.

1. V. Berzins and M. Gray, "Analysis and Design in MSG.84: Formalizing Functional Specifications", *IEEE Trans. on Software Eng. SE-11*, 8 (Aug. 1985).
2. V. Berzins, M. Gray and D. Naumann, "Abstraction-Based Software Development", *Comm. of the ACM 29*, 5 (May 1986), 402-415.
3. V. Berzins and Luqi, *The Semantics of Inheritance in Spec*, Computer Science, Naval Postgraduate School, 1987. NPS 52-87-032.
4. V. Berzins and Luqi, "Languages for Specification, Design and Prototyping", in *Handbook of Computer-Aided Software Engineering*, Van Nostrand Reinhold, 1988.
5. V. Berzins and Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada*, Addison-Wesley, 1988.
6. D. Bjoerner and C. Jones, *Formal Specification and Software Development*, Prentice Hall, Englewood Cliffs, NJ, 1982.
7. B. Cohen, W. T. Harwood and M. I. Jackson, *The Specification of Complex Systems*, Addison Wesley, Reading, MA, 1986.
8. "Ada Programming Language", American National Standards Institute/MIL-STD-1815A, DoD, 1983.
9. J. A. Goguen, J. W. Thatcher, E. G. Wagner and J. B. Wright, "Abstract Data Types as Initial Algebras and the Correctness of Data Representations", in *Proc. Conf. on Computer Graphics, Pattern Recognition, and Data Structures*, 1975, 89-93.
10. A. Goldberg and D. Robinson, *Smalltalk-80: The Language and its Implementation*, Addison Wesley, Reading, MA, 1983.
11. J. V. Guttag, E. Horowitz and D. R. Musser, "Abstract Data Types and Software Validation", *Comm. of the ACM 21*, 12 (1978).
12. J. V. Guttag and J. J. Horning, "A Larch Shared Language Handbook", *Science of Computer Programming 6* (1986), 135-157.
13. J. V. Guttag and J. J. Horning, "Report on the Larch Shared Language", *Science of Computer Programming 6* (1986), 103-134.
14. M. Herlihy and B. H. Liskov, "A Value Transmission Method for Abstract Data Types", *Trans. Prog. Lang and Systems 4*, 4 (Oct. 1982), 527-551.
15. C. Hewitt and H. Baker, "Actors and Continuous Functionals", in *Formal Description of Programming Concepts*, North-Holland, New York, 1978, 367-387.
16. C. A. R. Hoare, "Proof of Correctness of Data Representations", *Acta Informatica 1*, 4 (1972), 271-281.
17. L. Johnson, "Overview of the Knowledge-Based Specification Assistant", in *Proc. Second Annual RADC Knowledge-based Assistant Conference*, RADC(COES), Grifiss AFB, NY, 1987.
18. B. H. Liskov, A. Snyder, R. Atkinson and J. C. Schaffert, "Abstraction Mechanisms in CLU", *Comm. of the ACM 20*, 8 (Aug. 1977), 564-576.
19. B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. Schaffert and R. Scheifler, *CLU Reference Manual*, Springer Verlag, 1981.
20. Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *to appear in IEEE TSE*, 1988.
21. R. Nakajima, "IOTA", *IEEE Trans. on Software Eng.*, Feb. 1985.
22. M. Shaw, W. A. Wulf and R. L. London, "Abstraction and Verification in ALPHARD: Defining and Specifying Iteration and Generators", *Comm. of the ACM 20*, 8 (Aug. 1977), 553-564.
23. M. Shaw, *Alphard: Form and Content*, Springer Verlag, 1981.
24. W. Turski and T. Maibaum, *The Specification of Computer Programs*, Addison Wesley, Reading, MA, 1987.
25. W. A. Wulf, R. L. London and M. Shaw, "An Introduction to the Construction and Verification of Alphard Programs", *IEEE Trans. on Software Eng. SE-2*, 4 (Dec. 1976), 253-265.
26. A. Yonezawa, "Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics", Ph. D. Thesis, MIT, 1977.